Universidade Federal de Pernambuco
Graduação em Ciência da Computação
Centro de Informática

Trabalho de Graduação

MidasX: Um Framework de Middleware Baseado em Arquitetura de Software

Aluno: Osmany Barros de Freitas obf@cin.ufpe.br

Orientador: Nelson Souto Rosa nsr@cin.ufpe.br

Resumo

A proposta deste trabalho é estender o *framework* Midas, adicionando a noção de componente e conector abstrato e construindo um ambiente gráfico que permita o desenvolvimento semi-automático de sistemas de *middleware*. Nesse ambiente gráfico estarão disponíveis todos os elementos do *framework* para construção de *middleware*. Com tais elementos, o usuário poderá montar graficamente a configuração que deseja para o seu *middleware* e, através desta representação gráfica, gerar um script que instancia os elementos do Midas.

Agradecimentos

Tudo que sou e conquistei até aqui foi graças à grande misericórdia de Deus sobre minha vida. Sou grato pelas experiências que Ele tem me proporcionado viver e a condição de servi-lo como Seu filho.

Aos meus pais Marcos e Agneide, por todo amor que tenho deles, pelo esforço e empenho na minha educação. Pela valentia com que todas as lutas foram enfrentadas e vencidas por amor de mim e meu irmão. Por nunca faltar apoio, incentivo, conselho, por serem, acima de tudo AMIGOS. Ao meu irmão Ítallo, pelos momentos vividos juntos, pela companhia, pela força em tantas ocasiões.

A Nelson Rosa, meu orientador, que ao longo de todo trabalho de pesquisa exerceu muito mais do que seu papel como orientador, um grande companheiro.

A meus avós que com muito prazer cuidaram de mim, poupando, muitas vezes, duas noites de descanço. Aos meus tios, primos, por todo convívio e carinho de vocês.

A Joyce Figueiredo (Amor), por todo carinho, pela sua presença em minha vida, pela força em tantos momentos difíceis e pelas alegrias compartilhadas.

A minha primeira professora Deane, quando com um ano e 11 meses de vida tive minha primeira aula. A Dilma, que me alfabetizou e aos demais professores que, ao longo da minha vida, contribuíram para a minha formação.

A Andrea Queiroz, minha primeira professora de jovens. Guardo com muita alegria as aulas que tive, foram muito importantes para mim.

A todos que me receberam de braços abertos aqui em Recife: Ernande, Célia, Charles, Valéria, Geraldo, Neildo, Edmilson, Michel, Thiago, Thomaz, Kinho, Nice, Joana. A todos que não pouparam esforços em orações a meu favor, que Deus os abençoe.

A meus companheiros de TI: Nandinho e Diógenes, grande abraço.

Finalmente a todos os demais que participaram dessa importante conquista em minha vida vizinhos e colegas de faculdade.

Obrigado a todos vocês!!

Índice

1 INTRODUÇÃO 1

1.1 CARACTERIZAÇÃO DO PROBLEMA 1
1.2 OBJETIVOS E CONTRIBUIÇÕES DO TRABALHO 2
1.3 ESTRUTURA DO TRABALHO 2

2 CONCEITOS BÁSICOS 1

2.1 *MIDDLEWARE* 1
2.1.1 Modelos de Middleware *2*
2.2 *FRAMEWORK* 4
2.3 ARQUITETURA DE SOFTWARE 5
2.3.1 Componentes *5*
2.3.2 Conector *6*
2.3.3 Configuração *6*
2.4 JMX 6
Nível de Serviços Distribuídos *7*
Nível Agente *7*
Nível de Instrumentação *7*
2.5 CONSIDERAÇÕES FINAIS 7

3 TRABALHOS RELACIONADOS 9

3.1 INTRODUÇÃO 9
3.2 POLYORB 10
3.3 TAO (THE ACE ORB) 10
3.4 QUARTERWARE 11
3.5 ARCADEMIS 11
3.6 ZDUN 12
3.7 ASTER 12
3.8 CONSIDERAÇÕES FINAIS 12

4 MIDAS *FRAMEWORK* 13

4.1 ELEMENTOS ARQUITETURAIS 13
4.2 COMPONENTES 14
4.3 CONECTORES 14
4.4 IMPLEMENTAÇÃO DOS ELEMENTOS DO *FRAMEWORK* 15
4.4.1 Implementação dos Componentes *15*
4.4.2 Implementação dos Conectores *16*
4.4.3 Implementação do Configurador *17*
4.4.4 Implementação do Ambiente Gráfico – MidasX *19*
4.5 CONSIDERAÇÕES FINAIS 20

5 ESTUDO DE CASO 22

5.1 DEFINIÇÃO DA APLICAÇÃO 22
5.2 CRIAÇÃO DA CONFIGURAÇÃO NO MIDASX 23

5.3 INSTANCIAÇÃO DO MIDDLEWARE ATRAVÉS DO MIDAS 24
5.4 SERVIDOR 24
5.5 CLIENTE 25
5.6 CONSIDERAÇÕES FINAIS 25

6 CONCLUSÃO E TRABALHOS FUTUROS 26

6.1 CONCLUSÃO 26
6.2 TRABALHOS FUTUROS 27

Lista de Figuras

FIGURA 2.1– VISÃO DO PAPEL DO MIDDLEWARE 2
FIGURA 2.2– MONITOR DE TRANSAÇÃO DE PROCESSO 2
FIGURA 2.3– MIDDLEWARE RPC 3
FIGURA 2.4– MIDDLEWARE ORIENTADO A MENSAGEM 3
FIGURA 2.5– MIDDLEWARE ORB 4
FIGURA 2.6– DESENVOLVIMENTO BASEADO EM *FRAMEWORK.* 5
FIGURA 2.7– ARQUITETURA DO JMX 6
FIGURA 4.1– DESENVOLVIMENTO DO MIDAS 14
FIGURA 4.2– REPRESENTAÇÃO GRÁFICA DE UM COMPONENTE 14
FIGURA 4.3– REPRESENTAÇÃO GRÁFICA DE UM CONECTOR 15
FIGURA 4.4 – ARQUITETURA DOS COMPONENTES 15
FIGURA 4.5– MÉTODOS DA CLASSE ABSTRATA COMPONENT 16
FIGURA 4.6– ARQUITETURA DOS CONECTORES 17
FIGURA 4.7– CONECTOR COMPOSTO 17
FIGURA 4.8– CONFIGURADOR E ELEMENTOS GERENCIÁVEIS 18
FIGURA 4.9– ARQUIVO DE CONFIGURAÇÃO XML 18
FIGURA 4.10– EXEMPLOS DE *OBJECTNAME* 18
FIGURA 4.11 – ASSINATURA DE UM CONECTOR ABSTRATO 19
FIGURA 4.12– MIDASX 20
FIGURA 4.13– MENSAGEM DE CRIAÇÃO DO ARQUIVO XML 20

CAPÍTULO

1

Introdução

"Procure ser um homem de valor em vez de ser um homem de sucesso".

Albert Einsten.

A necessidade crescente de integração de aplicações desenvolvidas em rede associada ao aumento na complexidade destas aplicações tem levado ao uso extensivo de *middleware*. A natureza das aplicações que têm sido integradas é muito variada, o que inclui aplicações executando em nós de redes de sensores sem-fio, passa por aplicações que realizam computação em grade e chega às aplicações comerciais com grande volume de transações. Além disto, estas aplicações têm cada vez mais requisitos de segurança, tempo real e QoS (*Quality of Service*) que precisam ser atendidos por aplicações e por elementos de comunicação.

1.1 Caracterização do Problema

Comum a todas as aplicações mencionadas está a necessidade de diminuir o custo e a complexidade de desenvolvê-las. O *middleware* agiliza o desenvolvimento de aplicações distribuídas [3] e esta é a *razão* deste tipo de software. Mais especificamente, o middleware fornece um guia de implementação das aplicações e funciona como um *framework*, onde diversos serviços já estão implementados (e podem ser reusados) e o desenvolvedor pode se concentrar apenas em questões ligadas a funcionalidade da aplicação. Além disto, o fato de que aplicações distribuídas executam normalmente em ambientes heterogêneos, i.e., heterogeneidade de hardware, sistema operacional, ambientes de rede e linguagens de programação, torna o desenvolvimento deste tipo de aplicação mais complexo. Neste ponto, o

middleware trata esta heterogeneidade e permite a integração de aplicações nos mais diversos ambientes.

Apesar da popularidade de uso de middleware, a visão do desenvolvedor deste software restringe-se normalmente ao conhecimento das API's (*Application Programming Interface*) fornecidas para que ele/ela faça uso dos serviços e facilidades fornecidas pelo *middleware*. Na prática, o *middleware* é uma caixa-preta para os desenvolvedores de aplicações. De fato, passar a entender detalhes internos do funcionamento de um middleware é uma tarefa muito trabalhosa. Se considerarmos a possibilidade de construir um middleware, esta tarefa é de uma complexidade ainda muito maior. Logo, a despeito das facilidades para a construção de aplicações distribuídas, o entendimento/ desenvolvimento de um software básico como um middleware é uma tarefa complexa.

1.2 Objetivos e Contribuições do Trabalho

Neste trabalho apresentamos o *framework* MIDAS (*Middleware Development through Architectural abStractions*) para a construção de diversos sistemas de *middleware* a partir de um conjunto dos componentes básicos definidos no *framework*. Os componentes básicos são definidos adotando-se princípios de arquitetura de software [11]. Na prática, através do *framework* é possível desenvolver não apenas um *middleware*, mas diversos sistemas *middleware* que podem ser diferenciados de acordo com a composição dos componentes básicos utilizados. Logo, pretende-se que a construção do *middleware* possa ser realizada de forma rápida, ao mesmo tempo em que customizá-lo de acordo com requisitos da aplicação ou da infra-estrutura de rede disponível passe a ser uma tarefa mais fácil.

1.3 Estrutura do Trabalho

Este trabalho está estruturado da seguinte forma: a Seção 2 introduz os conceitos básicos *middleware*, *framework*, Arquitetura de software e *JMX*. Em seguida, na Seção 3, apresentamos o *framework* proposto. Para validá-lo, ele será usado para a construção de um *middleware* com invocação implícita. Na Seção 5 apresentamos alguns trabalhos relacionados. Finalmente, na Seção 6 são apresentadas as conclusões e possibilidades de trabalhos futuros.

CAPÍTULO

2

Conceitos Básicos

"O sucesso é conseqüência".

Albert Einsten.

Neste capítulo apresentamos os conceitos básicos necessários ao entendimento do *framework* proposto. Inicialmente são apresentados conceitos básicos de *middleware*, seguido por uma introdução de elementos relacionados à própria definição de *framework*. A seguir, uma vez que o *framework* de *middleware* proposto é definido em termos de elementos de arquiteturais, são introduzidos os principais conceitos de arquitetura de software.

2.1 *Middleware*

Como mencionado anteriormente, *middleware* [1][7][3] é um software de comunicação situado entre aplicação e o sistema operacional (ver Figura 2.1) com a finalidade de esconder detalhes de baixo nível do sistema operacional (comunicação e concorrência) e tratar a heterogeneidade de linguagens de programação, sistema operacional, redes e hardware.

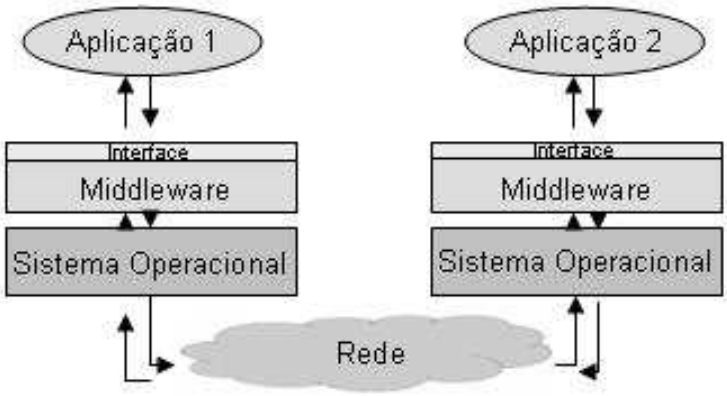

Figura 2.1– Visão do papel do middleware

Os serviços providos pelo *middleware* permitem a realização de uma comunicação transparente entre as aplicações distribuídas através de interfaces de alto nível que permitem aos programadores abstrair-se de diversos problemas inerentes às aplicações distribuídas.

Não há restrições para utilização do *middleware*, podendo ser aplicado em sistemas com dispositivos móveis, aplicações embarcadas, ou mesmo redes de sensores.

2.1.1 Modelos de *Middleware*

A maioria dos sistemas de *middleware* é baseada nos modelos, ou paradigmas, de comunicação: Processamento de Transações (TPs), Execução Remota de Procedimentos (RPCs), *Middleware* Orientado a Mensagem (MOMs), Intermediadores de Requisições de Objetos (ORBs).

Midleware Transacional

Os sistemas de middleware transacionais provêem aos sistemas cliente/servidor distribuído a capacidade de desenvolver, gerenciar e executar aplicações com transações de forma eficiente e confiável. Sendo assim, este modelo é bastante utilizado para realizações de negócios e atualizações em banco de dados.

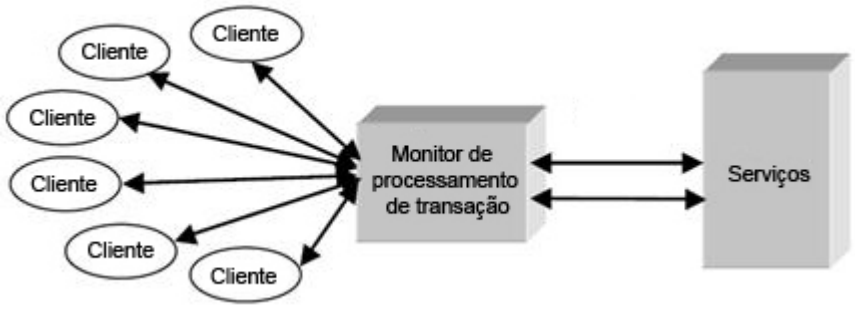

Figura 2.2– Monitor de Transação de Processo

Como visto na Figura 2.2, middleware transacional disponibiliza serviços de aplicações para milhares de clientes em um ambiente

cliente/servidor distribuído. Para essa funcionalidade, são realizadas multiplexações dos pedidos de transações dos clientes.

Middleware baseado em RPC

Define um protocolo para execução remota em computadores ligados em rede. O RPC pode ser implementado em diferentes protocolos de transporte. Por sua vez, não cabe ao RPC definir como a mensagem é enviada, mas apenas especificá-la e interpretá-la.

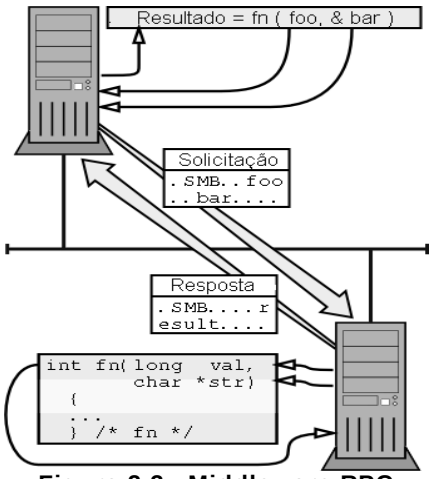

Figura 2.3– Middleware RPC

Middleware Orientado a Mensagem (MOM)

Um MOM aumenta a interoperabilidade, portabilidade e flexibilidade de uma aplicação, permitindo que a aplicação seja distribuída sobre múltiplas plataformas heterogêneas. O MOM reside em ambas as partes da arquitetura (Figura 2.4) e tem como característica o suporte de chamadas assíncronas entre o cliente e os servidores de aplicações.

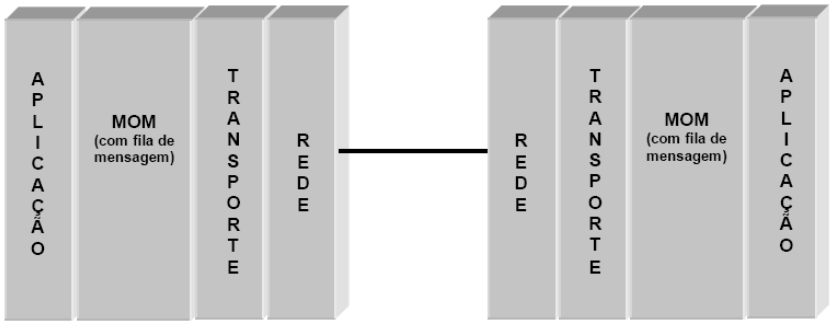

Figura 2.4– Middleware orientado a mensagem

O MOM tem por característica substituir a comunicação direta entre as aplicações por um mecanismo de troca de mensagens. As aplicações, portanto, se comunicam através da inserção de mensagens em filas específicas. Ainda que a aplicação não esteja disponível para receber a mensagem, ela será mantida na fila da aplicação. O cliente tem garantia apenas de que a mensagem será inserida na fila da aplicação destino, pois a responsabilidade de remover a mensagem da fila é da própria aplicação e não mais do *middleware*.

Intermediador de Requisição de Objetos (ORBs)

O ORB (Object Request Broker) permite a comunicação de objetos entre diferentes máquinas, diferentes softwares e diferentes fornecedores. Através do ORB um diretório de serviços é disponibilizado e auxilia a estabelecer conexão entre clientes e estes serviços.

O ORB deve suportar muitas funções com o objetivo de fornecer transparência ao usuário do ORB. Portanto é de responsabilidade do ORB fornecer a transparência de localidade, ou seja, fazer com que um objeto remoto pareça no local para o cliente.

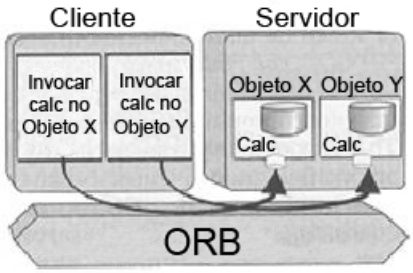

Figura 2.5– Middleware ORB

Na Figura 2.4 o cliente chama duas operações calc de objetos X e Y que estão localizados no servidor separado remotamente. Para o cliente, não importa saber onde estão os objetos X e Y, mas apenas que os resultados das suas requisições sejam obtidos.

2.2 *Framework*

Um *framework* provê uma solução para uma família de problemas semelhantes, onde as aplicações possuem em comum um mesmo tipo de problema [16]. O *framework* pode ser constituído por um conjunto de classes abstratas e concretas, bibliotecas de código ou ainda programas de apoio que colaboram para atingir um mesmo objetivo. Do ponto de vista prático, um *framework* é uma semi-aplicação flexível e extensível para permitir a elaboração de partes complementares específicas das aplicações.

O uso de um *framework* apresenta algumas vantagens: modularidade, pois permite encapsular implementações flexíveis em uma interface estável; reuso, uma vez que define elementos genéricos que podem ser reaproveitados; e extensibilidade, pois provê métodos que possibilitam às aplicações estenderem suas interfaces. Por outro lado, isso requer do desenvolvedor um conhecimento mais profundo no domínio abordado, pois manter a flexibilidade e ao mesmo

tempo permitir a eficiência das aplicações exige bastante atenção na construção do *framework*.

A Figura 2.5 mostra como funciona o processo de desenvolvimento das aplicações utilizando *framework.*

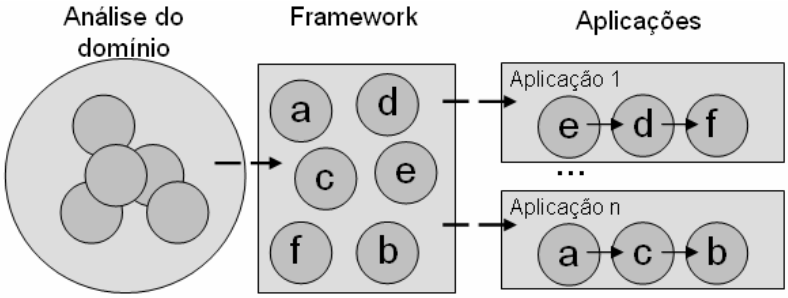

Figura 2.6– Desenvolvimento baseado em *framework.*

Na fase de análise do domínio é feita uma avaliação do problema a ser tratado para tornar conhecido seu domínio e aprofundar a experiência numa dada família de aplicações. O design do framework consiste de uma arquitetura elaborada com os elementos básicos necessários às aplicações mais comuns e que podem ser estendidos para aplicações mais específicas. Por fim, temos as aplicações já construídas com esses elementos compostos diferentemente de acordo com a necessidade de cada aplicação.

2.3 Arquitetura de Software

A arquitetura de software nasceu diante da necessidade de se identificar a estrutura de um sistema. Seu papel tem sido estabelecer um elo entre os requisitos e a implementação do sistema, dispondo de uma estrutura composta por elementos e seus relacionamentos. Tais estruturas têm sido desenvolvidas por linguagens específicas denominadas de Linguagens de Descrição de Arquitetura de software, conhecidas como ADLs.

As ADLs são capazes de descrever características estruturais e comportamentais dos sistemas, utilizando três elementos básicos: Componentes, Conectores e Configurações.

2.3.1 Componentes

São os elementos arquiteturais que possuem capacidade de processamento ou armazenamento de dados. Os componentes representam uma grande diversidade de elementos, desde simples funcionalidades até aplicações complexas. Esse elemento proporciona modularidade e separação de conceitos uma vez que escondem a implementação. Para permitir o acesso externo aos seus recursos é definida uma interface que pode ser utilizada por outros componentes. Tais interfaces definem as características funcionando como um contrato entre componentes, essas interfaces podem ser representadas, por exemplo, por: Portas, Pontos de Interação, Serviços.

2.3.2 Conector

Um conector é um elemento que representa o modelo de interação entre os componentes e define as regras que irão controlar essas interações. Concentram, portanto, as interconexões tornando-as visíveis e reusáveis. Semelhantemente aos componentes, os conectores podem possuir interfaces que contêm os pontos de interação entre o conector e o componente.

2.3.3 Configuração

Define a topologia do sistema, ou seja, os elementos que o constituem (Conectores e Componentes) e como estão interconectados formando arquitetura. Ou seja, a Configuração descreve como os componentes e conectores irão funcionar juntos no sistema.

2.4 JMX

A Sun desenvolveu uma tecnologia chamada JMX (*Java Management Extensios*) [11] para gerenciamento de código Java. A arquitetura do JMX permite que aplicações ou sistemas Java sejam gerenciados.

Com esta arquitetura é possível determinar quais elementos ou estruturas da aplicação estão funcionando, quais recursos estão sendo consumidos por cada uma delas, além de alterar valores de configuração. Essa gerência é suportada pela interface denominada *MBean*.

Dentro deste ambiente gerenciável os elementos da aplicação trocam informações através de notificações. A notificação é a forma utilizada para obter acesso as funcionalidades dos elementos que implementam a interface MBean.

A arquitetura do JMX é composta por três níveis: Distribuição, Agente, Instrumentação (Ver Figura 2.6).

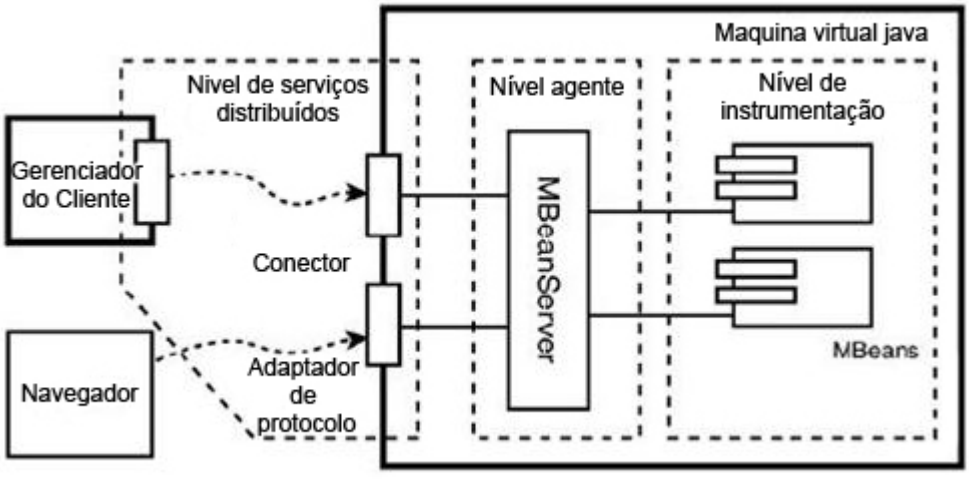

Figura 2.7– Arquitetura do JMX

A seguir uma breve descrição de cada um dos níveis.

Nível de Serviços Distribuídos

É a camada mais externa da arquitetura, cuja função é permitir externamente a disponibilidade dos elementos gerenciáveis, ou seja, do nível de instrumentação. Essa comunicação é proporcionada por dois elementos: Adaptador e Conector. Os Adaptadores promovem essa visibilidade através de componentes que implementam protocolos como HTTP e SNMP, sendo que tais componentes não existem nos clientes. Os conectores por sua vez utilizam tecnolgias distribuídas, como por exemplo, RMI.

Nível Agente

Provê acesso às entidades gerenciáveis pela aplicação oferecendo serviços como monitoramento, *timer*, carregamento dinâmico e serviços de relacionamento. É composto pelo *MBean Server* e um conjunto de serviços, tais como: Applet Gerenciável (M-Let), Monitoramento, Temporizador, Relacionamento.

O serviço M-Let, permite que um *MBean* seja monitorado mesmo que esteja rodando em qualquer lugar da rede.

Já o serviço de monitoramento realiza verificações sobre os atributos dos *MBeans*. Utilizando as variações nos atributos e condições pré-definidas, este serviço pode lançar notificações para que o sistema possa tratar comportamentos indesejados.

O Temporizador é capaz de enviar notificações em broadcast em intervalos de tempo e data definidos.

Por fim, o serviço de Relacionamento utiliza regras e tipos de relacionamento entre os *MBeans*. Com esse tipo de relacionamento é possível gerenciar conjuntos de *MBeans* automaticamente. Por exemplo, dada uma regra que defina que o *MBean* A só deve funcionar na existência do MBean B. Uma vez que B venha a ser desativado, este serviço se encarrega de desativar A preservando a regra pré-estabelecida.

Nível de Instrumentação

Define como podemos criar entidades gerenciáveis pelo JMX, i.e., os *MBeans*. Este nível consiste dos *MBeans* registrados no nível de agente. São eles que implementam os recursos gerenciáveis pelo sistema, cujas funcionalidades ou configurações são acessíveis por meio de interfaces.

2.5 Considerações Finais

Neste capítulo abordamos os principais conceitos necessários à compreensão deste trabalho. Inicialmente, definimos *Middleware* como software de comunicação que suporta a interoperabilidade de aplicações em plataformas heterogênes. Em seguida apresentamos as noções básicas de *Framework* como uma semi-aplicação flexível que fornece suporte ao desenvolvimento de outras aplicações. Os conceitos de arquitetura de software foram apresentados na seção seguinte. Por fim, vimos o JMX, uma tecnologia que permite o gerenciamento de aplicações Java, suportando o controle e

monitoramento remoto dos elementos gerenciáveis suportadas pela interface MBean.

CAPÍTULO

3

Trabalhos Relacionados

"Nunca diga às pessoas como fazer as coisas. Diga-lhes o que deve ser feito e elas surpreenderão você com sua engenhosidade."

George Patton.

Este capítulo apresenta alguns dos principais trabalhos relacionados à construção de sistemas de *middleware* com o uso de *frameworks.*

3.1 Introdução

Há diversas abordagens que propõem uma solução quanto ao desenvolvimento de sistemas de *middleware*. Estas abordagens podem ser agrupadas em categorias como as que seguem:

- Construir o *middleware* a "partir do zero": neste caso, os serviços fornecidos pelo *middleware*, o modelo de distribuição e os protocolos de comunicação precisam ser totalmente implementados. O projetista tem total liberdade de escolha de vários aspectos do middleware, mas fica responsável também por implementar tudo;

- Construir o *middleware* usando *framework* "caixa-branca": o projetista utiliza componentes e interfaces abstratas pré-definidas que devem ser implementados para a construção do middleware;

- Construir *middleware* usando *framework* "caixa-preta": componentes pré-definidos e já implementados são compostos ou estendidos pelo projetista para construção do *middleware*;

- Construir o *middleware* a partir de especificações de *interfaces* abstratas: neste caso o desenvolvedor implementa o *middleware* a partir de especificações fornecidas por empresas como a Sun (JMS – Java Message System, JTS - Java Transaction System) ou órgãos de padronização como a OMG (Object Management Group – CORBA [13]). Neste caso, o modelo de distribuição, os protocolos usados, a infra-estrutura de comunicação já estão definidos. A tarefa do projetista passa a ser a implementação, mas todas as decisões do que deve constar na implementação do *middleware* já foram tomadas;

- Construir serviços de middleware: neste caso os desenvolvedores reutilizam a infra-estrutura de comunicação e distribuição de middleware tradicionais como CORBA e RMI. Desta forma, o projetista fica com a tarefa de implementação de serviços comuns como segurança e transação ou de serviços específicos do domínio que podem se colocados no middleware.

Baseados nas estratégias apresentadas, os trabalhos a seguir concentram-se em apresentar estratégias detalhadas de implementação de middleware. Em função disto, a maioria deles não está interessada em apresentar apenas um produto final, mas sim de permitir a construção de famílias de middleware.

3.2 PolyORB

O foco principal do trabalho de Hughes [8] [8] é a prototipação rápida de *middleware* a partir de um conjunto de componentes básicos já implementados. A definição de componentes é possível a partir da constatação de que a maioria dos sistemas de *middleware* fornece mecanismos de tempo de execução e funcionalidades muito similares. Na prática, esta abordagem favorece o reuso de código por parte dos projetistas de *middleware*.

O *middleware* esquizofrênico proposto é uma arquitetura que consiste de componentes distribuídos em três partes: personalidades do nível de aplicação, núcleo e personalidades do nível de protocolo. As personalidades da aplicação funcionam como interfaces entre a aplicação e o núcleo e são responsáveis em traduzir uma chamada específica de uma aplicação em uma chamada independente de personalidade que pode ser processada pelo núcleo. As personalidades dos protocolos recebem as chamadas do núcleo e as traduzem em chamadas do protocolo de comunicação de baixo nível. Estes elementos são construídos sobre um conjunto de sete serviços básicos, cinco serviços avançados e foram desenvolvidos usando padrões de projetos.

3.3 TAO (The ACE ORB)

Na abordagem adotada por Schmidt [17] dois conceitos básicos são utilizados para o desenvolvimento de sistemas de *middleware*: padrões de projeto e *frameworks*. Os padrões de projeto reusam o conhecimento sobre um problema e sua solução para construção de software de um determinado domínio. Os *frameworks* são aplicações semi-completas implementadas a partir dos padrões de projeto. Desta forma, os padrões de projeto auxiliam a implementação dos *frameworks*, e estes por sua vez agilizam o desenvolvimento de um *middleware* ou de uma família de *middleware* [16].

É um *middleware* construído usando o framework ACE. O ACE por sua vez foi implementado a partir de um conjunto de padrões de projeto específicos para o domínio de *middleware*. Os padrões de projeto utilizados incluem: o Wrapper Facade para simplificar a interface de programação do sistema operacional; o Reactor usado para permitir a reação das aplicações a eventos do sistema operacional; o Acceptor-Connector usado para separar a etapa de estabelecimento da conexão das interações que ocorrem após esta etapa entre as aplicações; o padrão Leader Followers usado para controle de concorrência; o padrão Strategy para configuração do ORB no TAO; o padrão Abstract Factory para uso de múltiplas subclasses; o padrão Component Configurator para reconfiguração dinâmica do ORB; o padrão Proxy que define um objeto que funciona como um representante de outro objeto; e finalmente o padrão Adapter que permite diferentes objetos trabalharem em conjunto mesmo tendo diferentes interfaces.

Usando estes padrões de projeto, três *frameworks* foram então construídos. O *framework* Reactor que implementa os padrões de projeto Reactor e Leader Followers para auxiliar o controle de concorrência; o framework Acceptor –Connector implementa os padrões de projeto Accpetor-Connector e Strategy; e o framework Service Configurator implementa o padrão de projeto Configurator.

3.4 Quarterware

É um dos trabalhos pioneiros[18] em relação ao uso de técnicas de desenvolvimento de *middleware*. A proposta do Quarterware inclui a definição de uma metodologia, um *framework* e um toolkit de componentes usados no desenvolvimento de middleware. A metodologia, chamada de Software RISC (Reduced Instruction Set Computer), tem como base a decomposição funcional do *middleware* em componentes simples e a composição destes componentes de diversas formas para construção do middleware.

O *framework* proposto no Quarterware permite a implementação e a customização dinâmica dos sistemas de *middleware* em tempo de execução. O *framework* é composto por implementações de objetos definidos como básicos e de interação. Os objetos básicos definidos incluem: Message Handlers, Thread Pools, Schedulers, Marshalers, Locks e Marshal Buffers. Em termos de objetos de interação, os objetos definidos são Marshalers, ClientCallTasks, ServiceTasks, CallData, CallEnvironments e Makers. Usando o framework, foram realizadas implementações do CORBA, MPI e RMI.

3.5 Arcademis

É um *framework* baseado em Java para a construção de sistemas de middleware. O Arcademis [14] é composto por um conjunto de classes e interfaces abstratas e concretas que podem ser customizadas pelos desenvolvedores para o desenvolvimento do middleware. Estas classes incluem os seguintes elementos: Chain of Invokers, Request Sender, Response Receiver, Discovery Agency, Remote Object Representation, Middleware Protocol, Serialization Protocol, Transport Protocol, Connector, Acceptor, Activator, Request Receiver, Response Sender, Scheduler, Skeleton, Chain of Dispatchers e Stub.

Como definido no Arcademis, o middleware construído a partir do *framework* possui alguns aspectos que podem ser configurados pelo

desenvolvedor: transporte de dados, estabelecimento da comunicação, protocolo do middleware, protocolo de serialização, semântica da invocação, referência remota de objetos, serviço de descoberta de serviços, protocolo de invocação, protocolo de despacho, e política de prioridade. Por exemplo, o desenvolvedor pode escolher um protocolo de middleware que quando implementado em uma rede sem-fio envie um número menor de mensagens, comparado ao mesmo middleware em uma rede cabeada. O Arcademis foi aplicado na construção de RMI em J2ME.

3.6 Zdun

A proposta de Zdun [19] consiste na definição de uma linguagem de padrões que pode ser usada por desenvolvedores para projetar, implementar, estender, integrar, customizar ou construir middleware. A linguagem de padrões é definida a partir da combinação de padrões de projeto usados para construção do middleware. Os padrões definidos incluem o padrão composto broker (requestor, marshaller, invoker), padrões de gerenciamento do ciclo de vida (static instance, per-request instance e client-dependent instance), padrões de gerenciamento de recursos (leasing, lazy acquisition, pooling e passivation), padrões de extensão, padrões de infra-estrutura e padrões de invocação assíncrona. A linguagem de padrão de projeto definida por Zdun tem sido usados no projeto de sistemas de middleware baseados em CORBA, Web Services e .NET.

3.7 Aster

O Projeto Aster [9] adota uma estratégia baseada em arquitetura de software e requisitos não-funcionais para a construção de arquiteturas de middleware. Na prática, é definida uma ADL (Architecture Description Language) [11] baseada em UML (Unified Modelling Language), uma ferramenta, e um repositório de descrições arquiteturais. A ADL é usada para descrever a arquitetura de middleware usando conceitos básicos de arquitetura de software como componentes, conectores e configurações. O repositório contém descrições arquiteturais de middleware que incluem os serviços fornecidos pelo middleware, padrões de composição destes serviços para atender requisitos não-funcionais. A ferramenta, por sua vez, é usada para gerar todas as composições possíveis de arquiteturas de software de middleware que satisfazem determinados requisitos não-funcionais. O projetista do middleware então escolhe a composição que satisfaça os requisitos não-funcionais exigidos pela aplicação.

Outros *frameworks* existentes para construção de sistemas de *middleware* não se referem a *middleware* de propósito geral como os mencionados acima, mas tais: *middleware* para computação pervasiva [10], middleware para rede de sensores sem-fio[2], middleware para computação em grade [5], middleware para ambientes de computação móvel [4].

3.8 Considerações Finais

Neste capítulo foram apresentados os principais frameworks para a cosntrução de *middleware*: PolyORB, Arcademis, e Áster.

CAPÍTULO

4

Midas *Framework*

"Os pequenos atos que se executam são melhores que todos aqueles grandes que se planejam".

George C. Marshall.

Este capítulo descreve o *framework* proposto, o Midas. Como mencionado anteriormente, esta proposta consiste em desenvolver um *framework* para *middleware* utilizando conceitos de arquitetura de *software* e possibilitando a flexibilidade do gerenciamento e monitoramento dos elementos do *framework*. O capítulo apresenta a definição dos elementos do *framework*, Componentes, Conectores, a implementação dos elementos arquiteturais e, por fim, o ambiente gráfico MidasX.

4.1 Elementos Arquiteturais

Visando prover flexibilidade arquitetural ao *framework* os conceitos de arquitetura de software foram trazidos para o contexto de sistemas distribuídos, de modo a contribuir com a solução proposta.

Inicialmente foram definidos os elementos canônicos do *framework* (ver Figura 4.1) que servem como base para o desenvolvimento dos sistemas de *middleware*. Para isso, o desenvolvedor reusa, estende ou sobrepõe tais elementos, montando a configuração para o *middleware* desejado. Esses elementos canônicos são gerenciados por um configurador que é responsável também pela comunicação entre os elementos de modo a respeitar a restrição arquitetural de que dois componentes se comunicam apenas através de um conector.

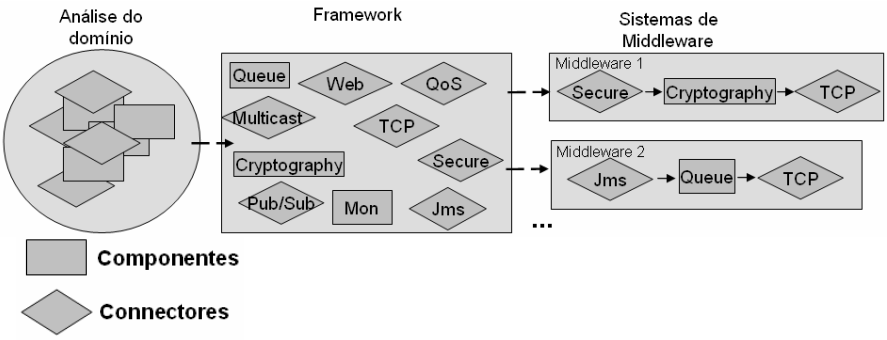

Figura 4.1– Desenvolvimento do MIDAS

De acordo com esta figura, uma vez definidos os elementos canônicos (Análise do Domínio), eles foram classificados em conectores e componentes de acordo com seu papel na arquitetura.

4.2 Componentes

Os componentes são os elementos do *framework* que implementam as funcionalidades básicas do *middleware* como segurança, criptografia, compressão, e a noção de fila. Cada componente pode prover um ou mais serviços externos, como também operações auxiliares internas. Para as funcionalidades acessíveis externamente é disponibilizado um conjunto de interfaces que permite o acesso aos recursos do componente. Estas interfaces de comunicação são implementadas utilizando o conceito de *Portas*, onde temos as portas de entrada e saída (ver Figura 4.2).

Figura 4.2– Representação gráfica de um Componente

Sempre que uma operação "provida" pelo componente precisa ser invocada, esta invocação é realizada na porta `invP` enviando-se os dados a serem processados. O resultado desse processamento é disponibilizado através da porta `terP`. Sempre que alguma solicitação é feita pelo componente e este necessita auxílio de operações de outros Componentes, ele "usa" uma operação externa. Nessa situação a porta de comunicação utilizada para invocar essa operação é a `invU` e o resultado é devolvido por meio da porta `terU`, prosseguindo assim a realização do processamento pelo Componente

4.3 Conectores

Os conectores não implementam funcionalidades como acontece com os Componentes, eles são elementos de comunicação entre os componentes. Conetores podem ainda ser combinaods para criação de novos conectores (Conectores Compostos).

De forma análoga aos componentes, os conectores (ver Figura 4.5) possuem portas que fornecem a comunicação do conector com os demais elementos. Cada conector possui uma lista de componentes conectáveis, ou

seja, elementos que podem se comunicar através deste conector. Essa lista é constituída por componentes cujas operações são compatíveis e estão contidas no contexto do conector em questão. Eles elementos compatíveis são definidos pelo desenvolvedor do conector.

Para os conectores, o funcionamento das portas assume outro padrão. Neste caso, toda invocação feita ao conector é acessada através da porta `invC` e seu resultado, após passar pelos componentes, é devolvido na porta `invS`. Quando há um retorno por parte da aplicação servidor para o cliente, essa resposta volta pela porta `terS`, realiza seus processamentos nos componentes e é despachada na porta `terC`.

Figura 4.3– Representação gráfica de um Conector

4.4 Implementação dos Elementos do *Framework*

Até o momento foi discutido como os conceitos de arquitetura de software foram utilizados para implementar os conectores e componentes, que compõem o núcleo do *framework,* denominados elementos canônicos. Os detalhes de implementação destes elementos canônicos serão apresentados a seguir.

4.4.1 Implementação dos Componentes

Uma vez que os componentes foram desenvolvidos, eles foram classificados e foi definida uma hierarquia buscando proporcionar melhor gerenciamento e controle dos componentes que serão instanciados (Ver Figura 4.3). Duas categorias foram então criadas: `Componentes Abstratos` e `Componentes Concreto`. Os `Componentes Concretos` implementam alguma funcionalidade de *middleware* ou armazenamento de dados diretamente e são, portanto, considerados elementos básicos. Por sua vez, os Componentes Abstratos são constituídos da composição de Componentes Concretos assumindo assim uma nova funcionalidade para o *middleware.*

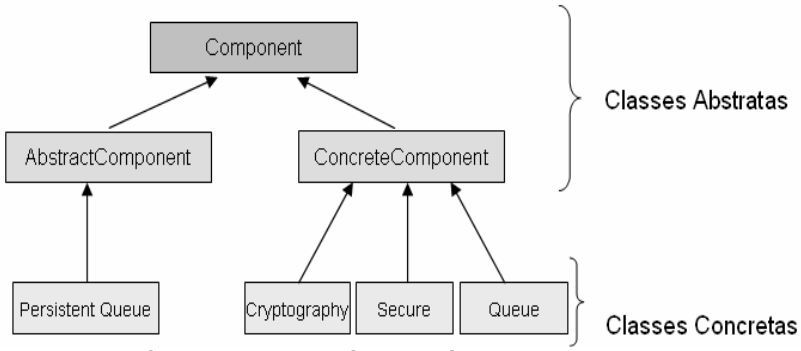

Figura 4.4 – Arquitetura dos Componentes

Em comum entre essas categorias existe a classe abstrata *Component* (Figura 4.4), ela define a assinatura das portas que serão adotadas por padrão a todos os demais componentes instanciados. Além de definir também alguns métodos gerais para funcionamento interno dos componentes. Esses métodos internos são para gerenciamento e processamento dos dados recebidos e enviados pelo componente.

```
(1) public void invP(final Message msg);
(2) public Message invU(int hashMsg);
(3) public Message terP(int hashMsg);
(4) public void terU(final Message msg);
(5) public String getCanonicalName();
(6) protected abstract void msgManager(Message msg);
(7) protected void sendNotification(Message msg, String port);
```

Figura 4.5– Métodos da classe abstrata Component

Como já mencionado anteriormente, as portas do componente (1)-(4) fornecem acesso aos métodos internos do componente e também permitem que o componente obtenha acesso externo à operações de outros elementos canônicos. Observe que toda essa interação é feita por meio da troca de uma classe *Message* (Seção 4.4.4) que possui os dados e todas as informações necessárias para o processamento dos componentes e movimentação ao longo dos conectores usados. Tais objetos são recebidos pelos Componentes e posteriormente acessados por meio de um código identificador para cada objeto.

Outro método utilizado e necessário para o funcionamento do *framework* é o que permite ao programador obter o nome do componente construído através classe abstrata Componente (*getCanonicalName*). Em seguida temos o método *msgManager* que é responsável pelo processamento dos dados recebidos, verificando as informações contidas no objeto *Message* e destinando às operações correspondentes. Finalmente, temos o método *SendNotification* que é responsável por comunicar ao configurador sempre que algum processamento for realizado dentro do Componente. Após essa notificação ao configurador, este acessa a interface do Componente e obtém o dado processado.

Cada uma das classes concretas (Figura 4.3) implementa funcionalidades básicas através de métodos internos acessíveis apenas através da porta `invC`.

Vale salientar que qualquer desenvolvedor que desejar implementar um componente ainda não disponível no *framework*, basta utilizar o padrão já definido e construir apenas o "*core*" do componente.

4.4.2 Implementação dos Conectores

Os conectores também estão estruturados hierarquicamente, de modo semelhante aos componentes como visto na Seção 4.3. Essa estrutura visa preservar o núcleo básico de processamento existente na classe Conector permitindo a flexibilidade necessária para os tipos derivados.

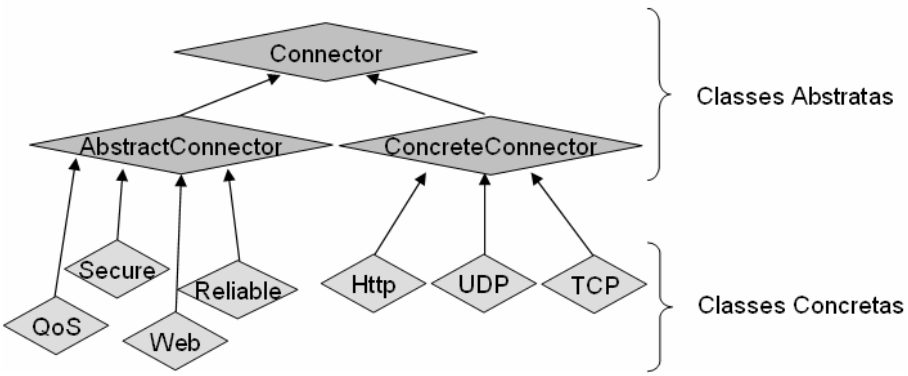

Figura 4.6– Arquitetura dos Conectores

Como podemos observar na Figura 4.6, temos duas categorias de conectores: `Conector Abstrato` e `Conector Concreto`. O `Conector Concreto` corresponde à conexão propriamente dita, ou seja, comunicação externa do *Middleware*, como "rtp", "http", "udp" e "tcp". São estes que estabelecem a conexão entre as aplicações suportando os protocolos de comunicação mais comuns. Por sua vez, o `Conector Abstrato` é construído pela composição de outros elementos canônicos. Este último é quem intermedia a comunicação entre componentes ou até mesmo entre conectores.

A responsabilidade dos conectores dentro do *framework* diz respeito à comunicação interna dos elementos canônicos e entre os sistemas de *middleware*. Portanto, cabe ao conector o papel de definir quais componentes serão utilizados, as regras de acesso para cada um deles e quais serviços serão solicitados em cada um desses componentes. Os conectores possuem ainda a flexibilidade de se unirem a outros conectores tendo em vista a criação do que denominamos `Conector Composto` - *ComposedConnector* integrando as funcionalidades dos conectores envolvidos.

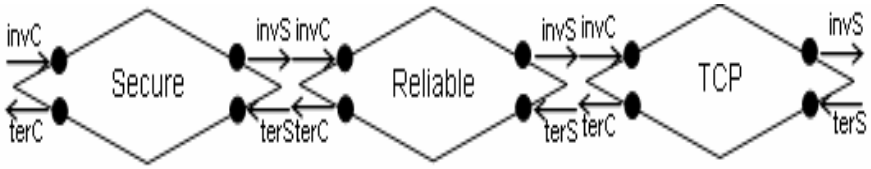

Figura 4.7– Conector Composto

A Figura 4.7 mostra a agregação de conectores para construção de um novo. Nesse exemplo, todo o processamento do primeiro conector é realizado pelos elementos que o compõem, em seguida os dados são transmitidos para o conector seguinte que por sua vez realiza suas operações. Por fim, o último conector recebe os dados processados e prepara-os para a transmissão destes pela rede.

4.4.3 Implementação do Configurador

Foi desenvolvido um elemento que utiliza a tecnologia JMX apresentada na Seção 2.4, de modo que as funcionalidades de gerenciamento e configuração dos elementos canônicos fossem disponibilizads no *framework*. Foi criado o `Configurador` que agrega não somente a função de configuração, mas de gerenciamento dos *MBeans*.

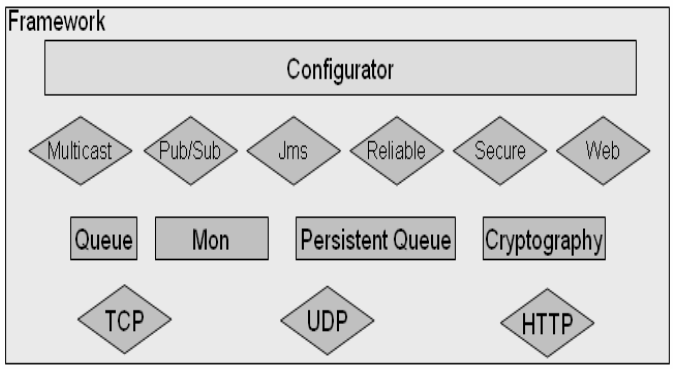

Figura 4.8– Configurador e elementos gerenciáveis

O `Configurador` assume um papel de meta-conector, uma vez que nenhum dos elementos canônicos possui comunicação interna direta. Isto significa que qualquer comunicação, seja conector-componente, componente-conector ou ainda conector-conector, é interceptada pelo `Configurador`. Dentro desta visão de gerenciamento suportada pelo JMX, todos os elementos gerenciáveis e configuráveis, os elementos canônicos, se comunicam através de notificações.

A configuração do *Middleware* é definida em um arquivo que determina quais elementos disponíveis no framework serão utilizados para instanciar o *Middleware* desejado.

```
<?xml version="1.0" encoding="UTF-8"?>
<Middleware id="1">
  <CanonicalElements>
    <Component id="Middleware:Type=Component,name=Cryptography" name="Cryptography" />
    <Component id="Middleware:Type=Component,name=Security" name="Security" />
    <Connector id="Middleware:Type=AbstractConnector,name=Reliable" name="Reliable" />
    <ConcreteConnector id="Middleware:Type=ConcretConnector,name=RTPClient" name="RTPClient" />
  </CanonicalElements>
  <Binds>
    <Bind id="1" to="Middleware:Type=Component,name=Security"
                 from="Middleware:Type=Component,name=Cryptography"
                 through="Middleware:Type=AbstractConnector,name=Reliable" />
  </Binds>
</Middleware>
```

Figura 4.9– Arquivo de configuração XML

A *tag CanonicalElements* (veja Figura 4.9) define todos os elementos canônicos que estarão disponíveis no *Middleware*: `Component, Connector e ConcreteConnector`. Cada um destes elementos possui dois atributos *id* e *name* que definem, nesta ordem, o *ObjectName* e o nome do MBean.

Todos os componentes e conectores estão registrados no `Configurador`. Dessa forma, qualquer notificação enviada por um elemento canônico é endereçada diretamente ao `Configurador`. Cada elemento canônico possui uma identificação própria para seu gerenciamento. Essa identificação é realizada através do *ObjectName* (estabelecido pela tecnologia JMX).

```
(1)  Middleware:Type=ConcretConnector,name=TcpServer
(2)  Middleware:Type=AbstractConnector,name=Reliable
(3)  Middleware:Type=Component,name=Cryptography
```

Figura 4.10– Exemplos de *ObjectName*

No Midas foi adotado um padrão para aquisição do *ObjectName* de um *MBean* (Figura 4.10), cuja composição é *Domínio*, *Chaves* e seus repectivos valores. Todos os elementos do *framework* estão no mesmo domínio *'Middleware'* e possuem a mesma chave. As chaves *'Type'* e *'Name'* definem o tipo e o nome do elemento, respectivamente.

Para que as operações do JMX sejam suportadas, é necessária a utilização de uma interface que defina o elemento como um *MBean.*

```
public class Reliable extends AbstracConnector
        implements ReliableMBean
```

Figura 4.11– Assinatura de um conector abstrato

A Figura 4.11 mostra como exemplo o conector *Reliable* que implementa uma interface *ReliableMBean*. Este simples recurso transforma o conector em um *MBean*. A classe *AbstractConnector* mostra a qual classificação dentro dos conectores está associada o conector *Reliable*.

As funções do configurador podem ser resumidas em:

- Acesso a todos os componentes e conectores existentes no *framework*;
- Recebimento das notificações de eventos gerados pelos nos elementos canônicos;
- Definir a comunicação interna do *framework*, gerenciando o tráfego dos dados pelos elementos gerenciáveis.

4.4.4 Implementação do Ambiente Gráfico – MidasX

Esse ambiente gráfico foi desenvolvido para facilitar a criação do arquivo de configuração do Midas. Fica pouco intuitivo para o usuário do Midas, conhecer os conectores e componentes disponíveis e criar à mão um arquivo XML que represente corretamente a arquitetura desejável.

O MidasX é um *plugin* para a plataforma de desenvolvimento Eclipse, construído com os recursos do *Graphical Editing Framework* (GEF). O GEF é um *framework* que permite a criação de editores gráficos baseado em modelos disponíveis.

Dentro do ambiente gráfico estão todos os elementos disponíveis do Midas, onde é possível configurá-los e exportar essa configuração em um arquivo XML.

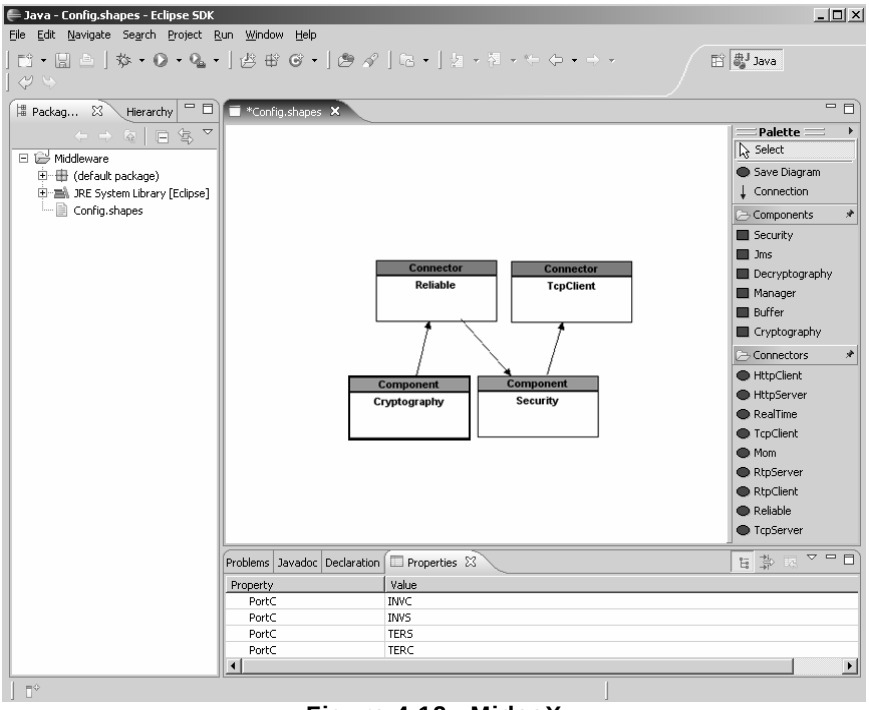

Figura 4.12– MidasX

A Figura 4.12 apresenta à direita as funcionalidades disponíveis no MidasX.

- Save Diagram: Gera o arquivo XML da configuração criada pelo usuário
- Connection: Estabelece a associação entre os elementos do *middleware*
- *Components* e *Connectors*: São os elementos disponíveis no Midas e que ficam à disposição do usuário para utilização.

Quando o usuário inicia o MidasX, os recursos são carregados automaticamente. A partir deste ponto, o usuário pode escolher quais elementos deseja utilizar em seu *Middleware* e vai adicionando ao editor. Uma vez definido como esses componentes e conectores irão se relacionar, é utilizada a ferramenta *Connection* para estabelecer esse relacionamento. Quando finalizada a configuração, basta clicar na ferramenta *Save Diagram* e o arquivo XML é então gerado (Figura 4.13).

Figura 4.13– Mensagem de criação do arquivo XML

4.5 Considerações Finais

Este capítulo apresentou os elementos e funcionamento do Midas e MidasX. Inciando pelos principais elementos arquiteturais da aborgadem do

Midas, como também o funcionamento básico dos componentes e conectores. Definindo como os elementos são acessados através das interfaces denomidas portas.

Foi introduzido na implementação dos elementos do *framework* os detalhes de funcionamento tanto para os Componentes quanto para os Conectores. Além da hierarquia das classes definidas e as classificações existentes.

A implementação do Configurador mostra quais as responsabilidades deste elemento, também chamado de meta-conector, por ser responsável por toda comunicação interna no Midas. O capítulo também propõe um modelo de arquivo de configuração a ser utilizado para instanciar sistemas de *middleware*.

Finalmente, a apresentação do ambiente gráfico – MidasX. Usado para facilitar a criação da cofiguração usada pelo Midas. Através do MidasX, o usuário ganha tempo e não precisa conhecer os detalhes que envolvem a geração de um arquivo de configuração.

CAPÍTULO

5

Estudo de Caso

"É preciso que você faça aquela coisa que acha que não pode fazer."

Eleanor Roosevelt.

Neste capítulo é apresentado o funcionamento do Midas utilizando como exemplo uma aplicação simples, mas que permite demonstrar o passo a passo de construção de um middleware e a utilização de seus recursos.

Inicialmente são apresentadas as características da aplicação que fará uso do middleware. Na sequência é mostrada a elaboração do perfil do *middleware* através do MidasX e o processo de instanciação através do Midas. Por fim, o funcionamento da aplicação sobre o *middleware* criado.

5.1 Definição da Aplicação

Buscando uma aplicação mais exigente no contexto de sistemas distribuídos, foi escolhido um exemplo de uma aplicação que opera com *stream* de vídeo sob arquitetura Cliente/Servidor.

Esta aplicação utiliza para a transmissão do vídeo o protocolo RTP – *Real Time Protocol*[20]. O RTP é utilizado em aplicações de tempo real, como por exemplo, distribuição de áudio fim a fim. Porém, o RTP também pode ser utilizado para comunicação com vários destinos, utilizando para isso um IP da faixa reservada para grupos *multicast*. Este protocolo tipicamente utiliza o UDP – *User Datagram Protocol* como protocolo de comunicação. Portanto, o RTP assume a característica de menor esforço e, consequentemente, não oferece

quaisquer garantias de que os pacotes serão entregues num determinado intervalo de tempo.

A aplicação cliente é um *player* implementado pela JMF – *Java Media Framework* para a exibição do vídeo recebido do servidor. JMF é uma API para manipulação de áudio, vídeo e outras mídias em aplicações Java. Sendo possível capturar *stream* de vídeo e áudio e codificá-lo em diversos formatos. O cliente também possui um *buffer* auxiliar que permite ao programador alterar seu tamanho.

5.2 Criação da configuração no MidasX

Com a aplicação definida, o passo seguinte é escolher quais elementos constituirão o *middleware* utilizado. Para este exemplo foram escolhidos o conector concreto RTP, o conector abstrato *RealTime* e o componente *Buffer*. O conector RTP implementa o protocolo necessário para a comunicação e transmisssão dos dados. Já o Componente *Buffer* recebe os pacotes do conector RTP e armazena os dados para disponibilizá-los à aplicação. O papel do conector *RealTime* é fazer essa comunicação entre o RTP e o Buffer.

Dentro do MidasX é iniciada a composição da arquitetura mencionada acima. Como pode ser visto na Figura 5.1, <<coloque aaqui um printscreen da ferramenta mesmo>> os elementos encontram-se disponíveis para utilização dentro do *pallete* e a estrutura de configuração está graficamente montada.

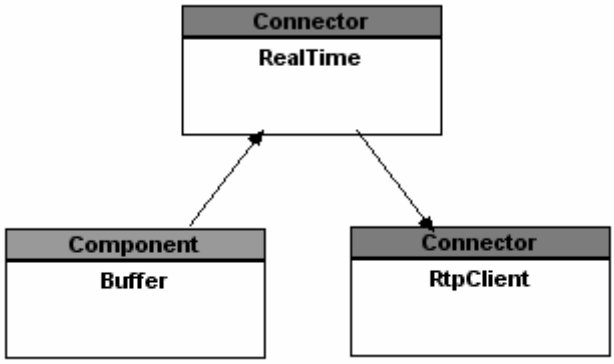

Figura 5.1 – Arquitetura do *middleware* no MidasX

O próximo passo é gerar o arquivo de configuração XML (Figura 5.2).

```
<?xml version="1.0" encoding="UTF-8"?>
<Middleware id="1">
  <CanonicalElements>
    <Component id="Middleware:Type=Component,name=Buffer" name="Buffer" />
    <Connector id="Middleware:Type=AbstractConnector,name=RealTime" name="RealTime" />
    <ConcreteConnector id="Middleware:Type=ConcretConnector,name=RTPClient" name="RTPClient" />
  </CanonicalElements>
  <Binds>
    <Bind id="1"
          to="Middleware:Type=Component,name=Buffer"
          from="Middleware:Type=Component,name=RealTime"
          through="Middleware:Type=AbstractConnector,name=RTPClient" />
  </Binds>
</Middleware>
```

Figura 5.2 – Arquivo de configuração

Essa etapa é encerrada e o arquivo de configuração deve ser utilizado pelo Midas.

5.3 Instanciação do middleware através do Midas

Com o arquivo de configuração criado (Seção 5.2), o usuário precisa apenas fornecê-lo como entrada na inicialização do Midas. Como não foi desenvolvido um servidor de nomes, a obtenção da interface remota está feita de modo explícito na aplicação. Portanto, para efeito de funcionamento, a obtenção da interface remota é realizada na própria aplicação, como será mostrado na seção seguinte.

5.4 Servidor

O servidor de vídeo é iniciado com o mesmo arquivo de configuração gerado para o cliente mostrado na Figura 5.2. O usuário clica na opção *Transmit File* (ver Figura 5.3) e escolhe a localização do vídeo, ip e porta (onde o cliente irá estabelecer a conexão). Essas entradas são enviadas ao conector RTP através dos elementos canônicos instanciados no *middleware*.

Figura 5.3 – Tela inicial do servidor

A Figura 5.4 apresenta mensagens que informam o estado de processamento do vídeo no servidor, contendo detalhes do vídeo escolhido, ip e porta disponíveis. A última mensagem *"Transmission setup OK"* indica que a transmissão está sendo realizada com sucesso.

```
MBeanServer Loaded
Processor configuring...
Processor configured.

Track #1 supports
MPEG/RTP
H263/RTP
JPEG/RTP
Track format set to MPEG/RTP

Realizing Processor...
Initialization successful for file:\D:\video.mpg
Started RTP session: 224.2.2.4 8088
Transmitting Track #1 ...

Transmission setup OK
```

Figura 5.4 – Janela de console do servidor

Quando os dados chegam ao conector RTP, este inicia o procedimento de transmissão do *stream* de vídeo

5.5 Cliente

Da mesma forma que o processo apresentado para inciar o servidor, a aplicação cliente, ou seja, o *player* de vídeo deverá receber como entrada o arquivo de configuração XML, a partir do qual seu *middleware* será instanciado. Na Figura 5.5 vemos a tela inicial do *player*.

Figura 5.5 – Janela de console do servidor

A aplicação então solicita o ip e porta do servidor. Esses dados são recebidos pelo *middleware* que encaminha ao conector RTP do cliente. O conector estabelece a conexão com o servidor e inicia então o recebimento do *stream* de vídeo.

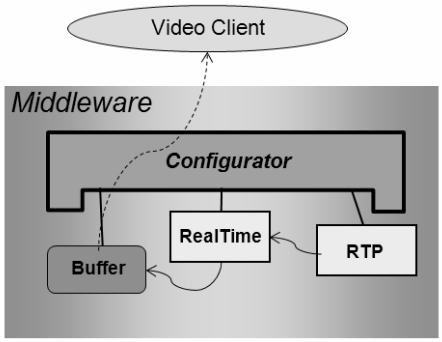

Figura 5.6 – Funcionamento básico do *middleware*

Na Figura 5.6 observamos o funcionamento interno do *middleware*. O conector RTP recebe os dados estruturados em datagramas e, de acordo com um buffer interno, notifica o *Configurator* informando que há dados sendo recebidos. O *Configurador* se encarrega de retirar os dados do conector RTP e levar ao componente *Buffer* que é responsável por armazenar os dados recebidos e enviar ao *player*. O *middleware* possui um temporizador que verifica a disponibilidade de dados para a sua aplicação, retira do *Buffer* e envia ao *player*, iniciando assim a exibição do vídeo.

5.6 Considerações Finais

Este capítulo mostrou um exemplo de utilização do Midas e MidasX em uma aplicação cliente/servidor. Foi abordado também a escolha da aplicação utilizada, a criação da arquitetura do *middleware* a ser instanciado, finalizando com a execução da aplicação utilizando do *middleware* gerado.

CAPÍTULO

6

Conclusão e Trabalhos Futuros

"Um grama de ação vale mais do que uma tonelada de teoria."

Friedrich Engels

Este capítulo apresenta as principais contribuições do trabalho e seus aspectos mais relevantes. As limitações dos modelos desenvolvidos e os trabalhos futuros a serem realizados dentro desta linha de pesquisa também são destacados.

6.1 Conclusão

Este trabalho apresentou o *framework* Midas para construção de sistemas de middleware e o MidasX, ambiente gráfico para configuração do Midas.

O Midas foi desenvolvido utilizando-se conceitos de arquitetura de software, tais como componentes e conectores. Esta abordagem facilita o reuso de elementos de comunicação (conectores) e de processamento (componentes) do *middleware*. Sua arquitetura define a estrutura básica para implementação de conectores e componentes. Com esse padrão pré-estabelecido, qualquer desenvolvedor pode criar novos elementos canônicos para incrementar o repositório já existente.

O MidasX tem como principal objetivo abstrair do usuário as preocupações e conhecimentos de implementação suficientes para construção da arquitetura do *middleware* desejado. Com esta ferramenta, tal atividade torna-se exclusivamente gráfica, representada por elementos visuais que facilitam o entendimento e agilizam a geração da configuração.

6.2 Trabalhos Futuros

Novos componentes e conectores que precisam ser desenvolvidos e adicionados ao Midas para que possam prover maior abrangência na construção de sistemas de *middleware*.

Há necessidade de aperfeiçoar o protocolo interno de comunicação usado pelo configurador para gerenciamento dos elementos. Atualmente, todos os dados necessários para identificar as notificações dos elementos canônicos são encapsulados em um único objeto. Otimizar esse funcionamento é fundamental para construção de Sistemas de *middleware* com mais elementos.

Em virtude da grande demanda de sistemas de middleware adaptativos, é interessante fornecer maior suporte a esse desenvolvimento. O objetivo é conceder autonomia ao *Configurator* para permitir troca de componentes e conectores em tempo de execução.

O MidasX poderá ser portado para o padrão RCP – *Rich Client Plataform* do ambiente de desenvolvimento Eclipse. Essa plataforma facilita a criação de aplicações *desktop* tornando o MidasX independente do Eclipse. Além do mais, o MidasX, quando acessar o repositório do Midas poderá verificar todas as restrições de comunicação entre conectores e componentes. Essa verificação, permitirá ao MidasX bloquear a elaboração visual de uma arquitetura estruturalmente incompatível.

Referências

[1] Bernstein, Philip A. (1996) “Middleware: A Model for Distributed System Services. Communications of the ACM”, Vol 39 (2), pp. 87-98, Fevereiro.

[2] Branch, J. W., Davis, J. S., Sow, D. M., e Bisdikian, C. Sentire (2005) “A Framework for Building Middleware for Sensor and Actuator Networks”. Third IEEE International Conference on Pervasive Computing and Communications, pp. 396 – 400.

[3] Campbell, Andrew T., Coulson, Geoff e Kounavis, Michael E. (1999) “Managing Complexity: Middleware Explained”, IT Professional, IEEE Computer Society, Vol 1(5), pp. 22-28, Outubro.

[4] Cepa, V., e Mezini, M. (2005) “A Generative Middleware Framework for Java Mobile Applications”. Proceedings of the 38th Annual Hawaii International Conference on System Sciences, pp. 283b - 283b, Janeiro.

[5] Coulson, G., Grace, P., Blair, G., Mathy, L., Duce, D., Cooper, C., Yeung, W.K., e Cai, W. (2004) “Towards a Component-based Middleware Framework for Configurable and Reconfigurable Grid Computing”. In Proceedings of the 13th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, pp. 291 – 296, Junho.

[6] Dumant, Bruno, Horn, François, Tran, Frédéric Dang e Stefani, Jean-Bernard (1999) “Jonathan: An Open Distributed Processing Environment in Java”. Distributed Systems Engeneering, Vol 6, pp. 3-12. Março.

[7] Emmerich, Wolfgang (2000) “Software Engineering and Middleware: A Roadmap”. Second International Workshop on Software Engineering and Middleware, Limerick. Ireland, pp. 119-129, Junho.

[8] Hugues, Jérôme, Pautet, Laurent, e Kordon, Fabrice (2003) “Contributions to Middleware Architectures to Prototype Distribution Infrastructures”. In Proceedings of the 14th IEEE International Workshop on Rapid Systems Prototyping (RSP’03), 2003, pp 1-8.

[9] Issarny, Valérie, Kloukinas, Christos, e Zarras, Apóstolos (2002) “Systematic Aid for Developing Middleware Architectures”. Communications of the ACM, Vol 45(6), pp. 53-58. Junho.

[10] Kumar, M. Shirazi, B. A. Das, S. K., Sung, B. Y., Levine, D., e Singhal, M. (2003) “PICO: a middleware framework for pervasive computing”. IEEE Pervasive Computing, Vol. 2(3), pp. 72 – 79.

[11] Lindfors, Juha “JMX – Managing J2EE with Java Management Extensions” , Sams Publishing, Janeiro 31, 2002.

[12] Medvidovic, Nenad e Taylor, Richard N. (2000) “A Classification and Comparison Framework for Software Architecture Description Languages”. IEEE Transactions on Software Engineering, Vol 26(1), pp. 70-93, Janeiro.

[13] OMG (2002) “Common Object Request Broker Architecture: Core Specification (CORBA 3.0)”, Dezembro.

[14] Pereira, F. M. Q., Valente, M. T. O., Bigonha, R. S. e Bigonha, M. A. S. (2004) “Arcademis: A Framework for Object Oriented Communication Middleware Development”. Software—Practice and Experience, pp. 1-18.

[15] Quinot, T., Kordon, F., Pautet, L. (2001) “From Functional to Architectural Analysis of a Middleware Supporting Interoperability across Heterogeneous Distribution Models”. In Proceedings of The 3rd International Symposium on Distributed Objects and Applications (DOA ‘01), pp. 165 – 175.

[16] Schmidt, D. C., e Buschmann, F. (2003) “Patterns, Frameworks, and Middleware: Their Synergistic Relationships”. Proceedings of 25th International Conference on Software Engineering, pp. 694 – 704, Maio.

[17] Schmidt, Douglas C. e Cleeland, Chris (1999) “Applying Patterns to Develop Extensible ORB Middleware”. IEEE Communications Magazine, Vol 37(4), pp. 54-63, Abril.

[18] Singhai, Ashish, Sane, Aamod, e Campbell, Roy H. (1998) “Quarterware for Middleware”. In Proceedings of The 18th International Conference on Distributed Computing Systems, pp. 192-201.

[19] Zdun, Uwe, Kircher, Völter, e Markus, Michael (2004) “Remoting Patterns Design Reuse of Distributed Object Middleware Solutions”. IEEE Internet Computing. Vol 8(6), pp. 60-68. Dezembro.

[20] July 31, 2007 RFC 4961 (*Symmetric RTP / RTP Control Protocol (RTCP)*) http://www.rfc-editor.org/rfc/rfc4961.txt